Clase 2 — 17.10.2025

#JAVA

👨‍🏫 *Profesor: Joan Salvador Gordi Ortega*
📘 Programación Multimedia y Dispositivos Móviles
🗓️ Clase 2 — 17/10/2025
🎯 Tema: **Repaso de la Programación Orientada a Objetos (POO) en Java**

1️⃣ Introducción a la Programación Orientada a Objetos (POO)

La **Programación Orientada a Objetos (POO)** surgió con el objetivo de **reutilizar código** y facilitar el **mantenimiento y la escalabilidad** del software.
Su auge se produjo en las décadas de **1980 y 1990** con lenguajes como **C++** y **Java**, que incorporaron este paradigma de forma nativa.

En la POO, **todo elemento del programa se modela como un objeto**, es decir, una representación abstracta de algo que existe en la realidad o que cumple una función dentro del sistema.

Ejemplo:

```
Lapiz lapiz01 = new Lapiz();
lapiz01.color = "rojo";
lapiz01.dibujar();
```

👉 **Objetivo principal:** modelar entidades del mundo real mediante **clases** que actúan como moldes para crear **objetos concretos (instancias).**

2️⃣ Clases y Objetos

Una **clase** es un **molde** que define qué propiedades y comportamientos tendrá un objeto.
Un **objeto** es una **instancia** concreta creada a partir de esa clase.

🔹 Propiedades

Son las **características** que describen un objeto.
Ejemplo: color, tamaño, material…

🔹 Métodos

Son las **acciones** que puede realizar el objeto.
Ejemplo: dibujar(), borrar(), mover()…

Ejemplo: Clase `Lapiz`

```
class Lapiz {
    String color;
    float grosor;
    String material;

    void dibujar() {
```

```
        System.out.println("Dibujando con un lápiz de color " + color);
    }

    void borrar() {
        System.out.println("Borrando con el lápiz");
    }
}
```

Crear instancias (objetos concretos)

```
Lapiz lapiz1 = new Lapiz();
lapiz1.color = "rojo";
lapiz1.dibujar();
```

🧩 **Palabra reservada `new`:** crea un nuevo objeto (instancia) en memoria.

📦 Cada instancia mantiene sus propias propiedades, independientes de otras.

3️⃣ Pilares Fundamentales de la POO

La POO se sustenta en **cuatro pilares**:

1. **Herencia**
2. **Encapsulación**
3. **Abstracción**
4. **Polimorfismo**

A continuación se desarrollan con ejemplos prácticos.

🗽Herencia

Permite **reutilizar código** al crear clases derivadas (subclases) que heredan atributos y métodos de una clase base (superclase).

Ejemplo:

```
class Persona {
    String nombre;
    String dni;
    String direccion;
    String fechaNacimiento;
}

class Alumno extends Persona {
    float notaAcceso;
}

class Profesor extends Persona {
    String titulacion;
}
```

- **Palabra clave:** `extends`
- **Ventaja:** evita duplicar código en clases similares.

🔸 **Ejemplo práctico:** un `Profesor` y un `Alumno` comparten propiedades comunes (nombre, DNI...) pero también tienen campos propios.

🗽Encapsulación

Su objetivo es **proteger los datos internos de un objeto** para que no puedan ser accedidos o modificados directamente desde el exterior.
De esta manera, se controla cómo se **leen y modifican** las propiedades, garantizando **seguridad, coherencia y mantenibilidad** en el código.

Una clase bien diseñada **no expone directamente sus atributos**. En su lugar, ofrece **métodos públicos controlados** para acceder o modificar esos valores.

Se logra con **modificadores de acceso** y **métodos Getters y Setters**.

Ejemplo1:

```
class Persona {
    private String dni;

    public void setDni(String dni) {
        if (dniValido(dni)) this.dni = dni;
        else System.out.println("DNI no válido");
    }

    public String getDni() {
        return dni;
    }

    private boolean dniValido(String dni) {
        // Validación del DNI
        return dni.matches("\\d{8}[A-Z]");
    }
}
```

- `private` → restringe el acceso directo.
- `public` → permite el acceso a través de métodos controlados.

✅ **Ventaja:** mantiene la integridad de los datos internos.

Ejemplo2:

Ejemplo conceptual:

```
class CuentaBancaria {
    private double saldo;  // Atributo privado

    public void depositar(double cantidad) {
        if (cantidad > 0) saldo += cantidad;  // Solo se acepta dinero positivo
    }

    public void retirar(double cantidad) {
        if (cantidad > 0 && cantidad <= saldo) saldo -= cantidad; // Control de saldo
```

```
    }

    public double consultarSaldo() {
        return saldo;
    }
}
```

🧩 En este ejemplo, el atributo `saldo` está protegido (`private`) y **solo puede modificarse a través de métodos** que validan las operaciones.

🔹 Modificadores de acceso

Modificador	Accesible desde la misma clase	Desde subclases	Desde el mismo paquete	Desde fuera del paquete
`private`	✅	❌	❌	❌
`protected`	✅	✅	✅	❌
`public`	✅	✅	✅	✅
(sin especificar) (*default*)	✅	❌	✅	❌

🔸 Por tanto, la encapsulación **usa** **`private`** para proteger los datos y **`public`** para los métodos que actúan como interfaz externa.

🔹 Getters y Setters

Los **Getters** y **Setters** son **métodos públicos** que permiten **acceder y modificar** atributos privados **de forma controlada**.

🧱 Getter

Sirve para **obtener** el valor de una propiedad privada.

Su nombre suele comenzar con `get` seguido del nombre del atributo con mayúscula inicial.

`public String getColor() { return color; }`

🧱 Setter

Sirve para **asignar** un valor al atributo.
Su nombre comienza con `set` .

`public void setColor(String nuevoColor) { color = nuevoColor; }`

👉 La clave es que en los **Setters** se puede **validar el valor recibido** antes de modificar la propiedad.

Ejemplo práctico

```
class Lapiz {
    private String color;
    private float grosor;

    // Getter
    public String getColor() {
        return color;
    }
```

```
    // Setter con validación
    public void setColor(String color) {
        if (color != null &&                        !color.isEmpty()) {
            this.color = color;
        } else {
            System.out.println("Color no                válido");
        }
    }

    public float getGrosor() {
        return grosor;
    }

    public void setGrosor(float grosor) {
        if (grosor > 0) {
            this.grosor = grosor;
        } else {
            System.out.println("El grosor               debe ser positivo");
        }
    }
}
```

Uso del objeto:

```
Lapiz lapiz = new Lapiz();
lapiz.setColor("Azul");        // Modifica el atributo de forma segura
lapiz.setGrosor(0.5f);
System.out.println(lapiz.getColor());  // Accede al valor
```

🔹 Beneficios de usar Getters y Setters

1. **Control de acceso:**
   Evita que el exterior modifique directamente valores sensibles.
   Ejemplo: impedir que el saldo de una cuenta sea negativo.
2. **Validación y seguridad:**
   Permiten verificar que los datos asignados cumplen reglas lógicas.
3. **Mantenimiento:**
   Si la lógica de acceso cambia (por ejemplo, agregar logs o encriptación), no afecta al resto del código que usa la clase.
4. **Compatibilidad:**
   Cambiar internamente el nombre o tipo del atributo no rompe el código externo, mientras mantengamos los métodos `get` y `set`.

🔹 Ejemplo avanzado: uso combinado con lógica de negocio

```
class Persona {
    private String dni;
    private int edad;

    public void setDni(String dni) {
        if (dni.matches("\\d{8}[A-Z]")) {
```

```
            this.dni = dni;
        } else {
            System.out.println("DNI                    inválido");
        }
    }

    public String getDni() {
        return dni.substring(0, 4) +               "****"; // Oculta parte del DNI
    }

    public void setEdad(int edad) {
        if (edad >= 0 && edad <= 120) {
            this.edad = edad;
        } else {
            System.out.println("Edad no                válida");
        }
    }

    public int getEdad() {
        return edad;
    }
}
```

Aquí el getter **devuelve el dato parcialmente oculto**, lo que refuerza el principio de **protección y privacidad** del objeto.

🗽 Abstracción

La **abstracción** consiste en **simplificar la complejidad del sistema** mostrando solo lo esencial y ocultando los detalles internos de funcionamiento.
En otras palabras, permite **centrarse en lo que un objeto hace**, no en **cómo lo hace**.

El usuario interactúa mediante **interfaces o métodos simples** sin conocer el funcionamiento interno.

Ejemplo:

```
class BusEscolar {
    void acelerar() {
        System.out.println("El autobús             acelera");
    }

    void frenar() {
        System.out.println("El autobús             frena");
    }
}
```

👁️ El conductor (usuario) no necesita conocer los mecanismos internos del motor o los pedales: solo usa los métodos públicos.

Cuando usamos un objeto, normalmente no necesitamos conocer su estructura interna ni cómo logra realizar sus tareas.

La abstracción permite que un usuario trabaje con **una interfaz clara y simple**, mientras los procesos complejos se mantienen ocultos dentro de la clase.

Ejemplo cotidiano:
Un conductor usa los pedales, el volante y los botones del coche sin conocer el funcionamiento del motor o del sistema eléctrico.

En POO, esos “controles” serían los **métodos públicos** de la clase.

```
class BusEscolar {
    // Atributos privados (ocultos)
    private int velocidad;
    private boolean motorEncendido;

    // Métodos públicos (interfaz de uso)
    public void encenderMotor() {
        motorEncendido = true;
        System.out.println("Motor                encendido.");
    }

    public void acelerar() {
        if (motorEncendido) {
            velocidad += 10;
            System.out.println("El bus                acelera a " + velocidad + "
km/h");
        } else {
            System.out.println("No se                puede acelerar. El motor está
apagado.");
        }
    }

    public void frenar() {
        if (velocidad > 0) {
            velocidad -= 10;
            System.out.println("El bus                reduce la velocidad a " +
velocidad + " km/h");
        } else {
            System.out.println("El bus ya                está detenido.");
        }
    }
}
```

En este ejemplo, el usuario **no necesita saber cómo se calcula la velocidad ni cómo se enciende el motor**, solo usa los métodos `encenderMotor()`, `acelerar()` y `frenar()`.

🔹 Diferencia con la encapsulación

- **Encapsulación:** protege los datos internos (oculta el *cómo se accede*).
- **Abstracción:** simplifica la interacción (oculta el *cómo funciona*).

Ambos conceptos trabajan juntos: la abstracción define la **interfaz pública**, y la encapsulación protege la **implementación interna**.

🔹 Ejemplo práctico con abstracción y encapsulación combinadas

```
class Cafetera {
    private int agua;  // encapsulado

    public void llenarAgua(int cantidad) {
        agua += cantidad;
    }

    public void prepararCafe() {   //          abstracción
        if (agua >= 100) {
            calentarAgua();
            molerGranos();
            System.out.println("☕ Café                 preparado");
            agua -= 100;
        } else {
            System.out.println("No hay                 suficiente agua");
        }
    }

    // Detalles internos ocultos
    private void calentarAgua() { /*           proceso interno */ }
    private void molerGranos() { /*            proceso interno */ }
}
```

El método `prepararCafe()` representa la **abstracción**, mientras que los métodos privados `calentarAgua()` y `molerGranos()` son parte de la **implementación oculta** (encapsulación).

🗽 Polimorfismo

El **polimorfismo** es el pilar de la POO que permite que **un mismo método o comportamiento adopte distintas formas** según el **tipo real del objeto** que lo utilice.

Proviene del griego *“poli”* (muchas) y *“morphé”* (formas), y es esencial para lograr **flexibilidad, reutilización y extensibilidad** en el código orientado a objetos.

Ejemplo:

```
class Profesor {
    float calcularSueldo(float sueldoBase,     float porcentajeRetencion) {
        return sueldoBase - (sueldoBase *          porcentajeRetencion / 100);
    }
}

class ProfesorNormal extends Profesor {
    @Override
    float calcularSueldo(float sueldoBase,     float porcentajeRetencion) {
        float valor = sueldoBase + 100; //          complemento
        return valor - (valor *                    porcentajeRetencion / 100);
    }
}

class ProfesorTutor extends Profesor {
    @Override
    float calcularSueldo(float sueldoBase,     float porcentajeRetencion) {
```

```
        float valor = sueldoBase + 200;
        return valor - (valor *                    porcentajeRetencion / 100);
    }
}
```

En Java, el polimorfismo permite **tratar objetos de distintas clases como si fueran de una clase común**, siempre que **hereden de ella** o **implementen una misma interfaz**.

Esto significa que una **referencia de tipo base** puede apuntar a **objetos de clases derivadas**, ejecutando el método que corresponda según su tipo real.

```
Persona p = new Profesor();     // p es una Persona, pero apunta a un Profesor
Persona q = new Alumno();       // q es una Persona, pero apunta a un Alumno
```

Cuando se llama a un método sobre estas referencias, Java ejecuta la versión del método correspondiente al **tipo real** del objeto, no al tipo de la referencia.

🧠 Esto se conoce como **ligadura dinámica** (*dynamic binding*) o **polimorfismo en tiempo de ejecución**.

@Override | final:

🧩 **@Override** → indica que un método se está **reescribiendo** (sobrescribiendo) en una subclase.

🧱 `final` evita la sobrescritura.

Ejemplo:

```
final float calcularSueldo(...) { ... }
```

```
Profesor p1 = new ProfesorNormal();
Profesor p2 = new ProfesorTutor();

System.out.println(p1.calcularSueldo(1000, 15)); // Usa lógica de ProfesorNormal
System.out.println(p2.calcularSueldo(1000, 15)); // Usa lógica de ProfesorTutor
```

El compilador reconoce el **tipo dinámico** del objeto y ejecuta el método correspondiente → **polimorfismo dinámico**.

🔹 Tipos de polimorfismo en Java

Tipo	Descripción	Momento de resolución
Sobrecarga (Overloading)	Mismo método con distinto número o tipo de parámetros dentro de la misma clase.	En **tiempo de compilación**
Sobrescritura (Overriding)	Redefinición de un método heredado para cambiar su comportamiento.	En **tiempo de ejecución**

Ejemplo 1: Sobrecarga (*Compile-time Polymorphism*)

```
class Calculadora {
    int sumar(int a, int b) {
        return a + b;
    }
```

```
    double sumar(double a, double b) {
        return a + b;
    }

    int sumar(int a, int b, int c) {
        return a + b + c;
    }
}
```

Aquí el método `sumar()` tiene **distintas firmas** (diferente tipo o cantidad de parámetros).
El compilador decide cuál ejecutar según los argumentos que se pasen:

```
Calculadora c = new Calculadora();
System.out.println(c.sumar(5, 2));       // Usa sumar(int, int)
System.out.println(c.sumar(3.5, 2.1));   // Usa sumar(double, double)
System.out.println(c.sumar(1, 2, 3));    // Usa sumar(int, int, int)
```

Ejemplo 2: Sobrescritura (*Runtime Polymorphism*)

```
class Profesor {
    float calcularSueldo(float sueldoBase,     float retencion) {
        return sueldoBase - (sueldoBase *          retencion / 100);
    }
}

class ProfesorNormal extends Profesor {
    @Override
    float calcularSueldo(float sueldoBase,     float retencion) {
        float total = sueldoBase + 100; //         bonificación
        return total - (total * retencion          / 100);
    }
}

class ProfesorTutor extends Profesor {
    @Override
    float calcularSueldo(float sueldoBase,     float retencion) {
        float total = sueldoBase + 200; //         bonificación mayor
        return total - (total * retencion          / 100);
    }
}
```

Uso del polimorfismo:

```
Profesor p1 = new ProfesorNormal();
Profesor p2 = new ProfesorTutor();

System.out.println(p1.calcularSueldo(1000, 15)); // Llama a versión de ProfesorNormal
System.out.println(p2.calcularSueldo(1000, 15)); // Llama a versión de ProfesorTutor
```

Aunque ambas variables son del tipo `Profesor`, cada una **ejecuta la versión correspondiente a su tipo real**, gracias al polimorfismo dinámico.

Ejemplo 3: Polimorfismo con interfaces

El polimorfismo también se aplica mediante **interfaces**, cuando distintas clases implementan los mismos métodos.

```
interface Animal {
    void hacerSonido();
}

class Perro implements Animal {
    public void hacerSonido() {
        System.out.println("Guau guau");
    }
}

class Gato implements Animal {
    public void hacerSonido() {
        System.out.println("Miau miau");
    }
}
```

Uso:

```
Animal a1 = new Perro();
Animal a2 = new Gato();

a1.hacerSonido();  // Guau guau
a2.hacerSonido();  // Miau miau
```

🔹 Aquí, la referencia `Animal` **no necesita conocer** qué tipo de animal representa: cada uno se comporta de forma distinta al llamar al mismo método.

Ejemplo 4: Aplicación práctica — Android

En Android, el método `onCreate()` se define en la clase `Activity` base, pero **cada actividad la sobrescribe** para implementar su comportamiento particular:

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle             savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        System.out.println("MainActivity           iniciada");
    }
}

public class LoginActivity extends Activity {
    @Override
    protected void onCreate(Bundle             savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
        System.out.println("LoginActivity          iniciada");
```

```
    }
}
```

El **sistema Android** llama a `onCreate()` sin importar el tipo de actividad, y cada clase **responde de forma diferente** → eso es polimorfismo en acción.

🔹 Beneficios del polimorfismo

1. **Código más flexible y escalable**
   Permite añadir nuevas clases sin modificar el código existente.
2. **Reutilización y mantenibilidad**
   El mismo método puede funcionar con distintos tipos de objetos.
3. **Desacoplamiento**
   El código depende de **interfaces o clases base**, no de implementaciones concretas.
4. **Extensibilidad en proyectos grandes (Android, Java EE, etc.)**
   En Android, muchas clases base (como `Activity`, `View`, `Adapter`) se comportan polimórficamente: el sistema ejecuta versiones personalizadas definidas por el desarrollador.

4️⃣ Clases Abstractas e Interfaces

🔹 Clases abstractas

Sirven como **plantillas** para otras clases.
No se pueden instanciar directamente y pueden contener métodos abstractos (sin implementación).

```
abstract class Persona {
    String nombre;
    abstract void caminar();
}
```

Una clase derivada **debe implementar** los métodos abstractos:

```
class Alumno extends Persona {
    @Override
    void caminar() {
        System.out.println(nombre + "              camina hacia clase");
    }
}
```

🔹 Interfaces

Definen un **contrato** que las clases que las implementan deben cumplir.
Solo contienen métodos abstractos (sin cuerpo) y constantes.

```
interface ICaminar {
    void caminar();
}
```

```
class Profesor implements ICaminar {
    public void caminar() {
```

```
        System.out.println("El profesor            camina hacia el aula");
    }
}
```

🧩 Una clase puede implementar **múltiples interfaces**, pero solo **heredar de una clase**.

5️⃣ Conclusiones

- La **POO** permite desarrollar software modular, mantenible y escalable.
- Las **clases** son los moldes y los **objetos** sus instancias.
- Los **pilares (herencia, encapsulación, abstracción, polimorfismo)** garantizan reutilización, seguridad y flexibilidad.
- El uso correcto de **getters/setters**, **@Override**, **final**, **abstract** e **interfaces** son herramientas clave para la arquitectura de Android y Java en general.
